D20A 1 Vanshika Ambwani Blockchain Lab

Exp 5

Aim: Deploying a Voting/Ballot Smart Contract

Theory:

1. Importance of ‘require’ Statements in Solidity

In Solidity, the require statement is used to validate conditions before executing critical parts of a smart contract. It acts as a safety checkpoint that ensures only legitimate inputs and authorized users can proceed with a function call. If the specified condition evaluates to false, the transaction is immediately reverted, and any changes made during execution are undone. This protects the contract from invalid operations and preserves blockchain integrity.
In a Voting (Ballot) smart contract, require statements can be applied to enforce rules such as:

- Verifying that a voter has the right to vote.
- Preventing a voter from casting multiple votes.
- Restricting certain functions (like granting voting rights) to the chairperson only.

Additionally, require allows developers to include descriptive error messages, which improve debugging and user interaction. Overall, it strengthens contract security, reliability, and correctness.

2. Key Solidity Concepts: mapping, storage, and memory mapping

A mapping in Solidity is a key-value data structure used to associate one type of data with another. Its syntax is:
mapping(keyType => valueType) For example:

mapping(address => Voter) public voters;

In this case, each Ethereum address is linked to a Voter struct containing relevant information such as voting weight, voting status, and selected proposal.
Mappings are highly efficient for data retrieval and are widely used in voting contracts. However, they do not maintain a length property and cannot be directly iterated over, making them efficient for lookups but limited for enumeration.

Storage

Storage refers to the permanent data area on the blockchain. State variables declared in a contract are stored in storage by default. Data stored here remains available across multiple transactions unless modified.
Because storage writes consume significant gas, developers must use it carefully. For example, voter details stored in a mapping remain permanently saved throughout the contract's lifecycle.

Memory

Memory is temporary storage used during function execution. Variables declared with the memory keyword exist only for the duration of the function call and are discarded afterward.
Memory is less expensive compared to storage and is suitable for temporary variables, calculations, or function parameters. For instance, temporary string handling or intermediate computations are typically done in memory.
Smart contract developers must carefully choose between storage and memory to optimize performance and reduce gas costs.

3. Why Use bytes32 Instead of string?

In earlier versions of the Ballot contract, proposal names were often stored as bytes32 instead of string.

- bytes32 is a fixed-size data type that stores exactly 32 bytes.
- It is gas-efficient, easier to compare, and simpler for the EVM to handle.
- However, it limits text length to 32 characters, reducing flexibility.

On the other hand:

- string is a dynamic data type that allows variable-length text.
- It improves readability and user-friendliness.
- However, it requires more complex storage handling and consumes more gas.

Therefore, bytes32 is preferred when efficiency and lower gas costs are priorities, while string is chosen when readability and flexibility are more important.

Code:

```
// SPDX-License-Identifier: GPL-3.0 pragma solidity ^0.8.0;

/**
*	@title Ballot
*	@dev Implements voting process along with vote delegation
*/
```

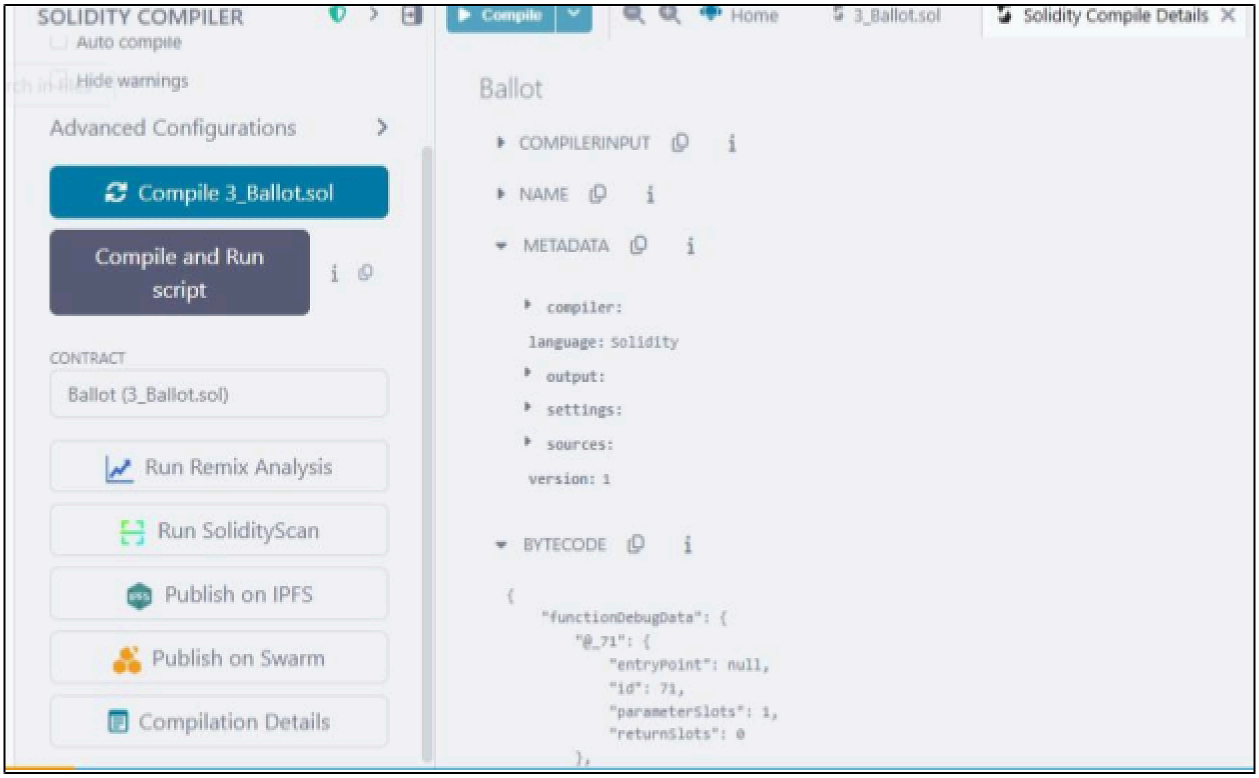
```
contract Ballot {

struct Voter {
uint256 weight;         // weight is accumulated by delegation bool voted;    // if true, that person
already voted address delegate; // person delegated to
uint256 vote;           // index of the voted proposal
}

struct Proposal {
string name;            // proposal name
uint256 voteCount; // number of accumulated votes
}

address public chairperson; mapping(address => Voter) public voters; Proposal[] public
proposals;

/**
* @dev Create a new ballot to choose one of 'proposalNames'.
*/
constructor(string[] memory proposalNames) { chairperson = msg.sender;
voters[chairperson].weight = 1;

for (uint256 i = 0; i < proposalNames.length; i++) { proposals.push(
Proposal({
name: proposalNames[i], voteCount: 0
})
);
}
}

/**
* @dev Give 'voter' the right to vote. Only chairperson can call.
*/
function giveRightToVote(address voter) external {
require(msg.sender == chairperson, "Only chairperson can give right to vote");
require(!voters[voter].voted, "The voter already voted"); require(voters[voter].weight == 0, "Voter
already has voting rights");

voters[voter].weight = 1

}

/**
* @dev Delegate your vote to another voter.
*/
function delegate(address to) external { Voter storage sender = voters[msg.sender];

require(sender.weight > 0, "You have no right to vote"); require(!sender.voted, "You already
voted");
require(to != msg.sender, "Self-delegation is not allowed");
```

```
// Follow the chain of delegation
while (voters[to].delegate != address(0)) { to = voters[to].delegate;
require(to != msg.sender, "Delegation loop detected");
}

Voter storage delegate_ = voters[to];
require(delegate_.weight > 0, "Delegate has no right to vote");

sender.voted = true; sender.delegate = to;

if (delegate_.voted) {
// If the delegate already voted, add directly proposals[delegate_.vote].voteCount +=
sender.weight;
} else {
// If the delegate did not vote yet, add weight delegate_.weight += sender.weight;
}
}

/**
* @dev Cast your vote.
*/
function vote(uint256 proposal) external { Voter storage sender = voters[msg.sender];

require(sender.weight > 0, "No right to vote"); require(!sender.voted, "Already voted");
require(proposal < proposals.length, "Invalid proposal index"); sender.voted = true

sender.vote = proposal; proposals[proposal].voteCount += sender.weight;

}

/**

* @dev Returns index of winning proposal.

*/

function winningProposal() public view returns (uint256 winningProposal_) { uint256

winningVoteCount = 0;

for (uint256 p = 0; p < proposals.length; p++) {

if (proposals[p].voteCount > winningVoteCount) { winningVoteCount = proposals[p].voteCount;

winningProposal_ = p;

}

}

}
```

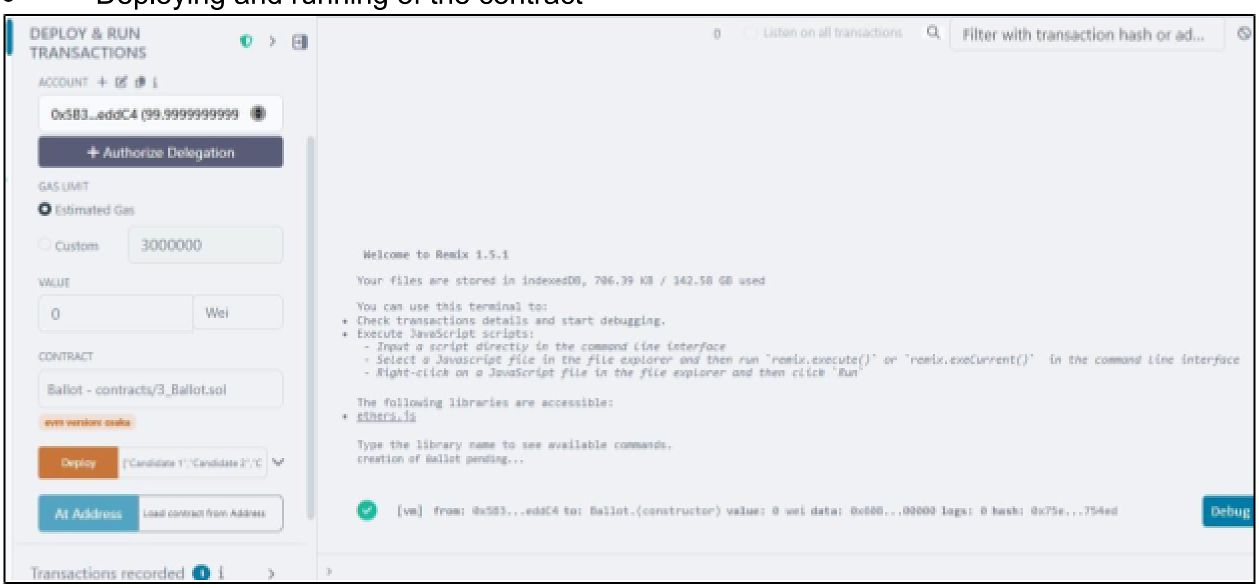
```
/**

* @dev Returns name of winning proposal.

*/

function winnerName() external view returns (string memory winnerName_) { winnerName_ =

proposals[winningProposal()].name;

}

}
```

Output:

- Compiled Ballot.sol Contract

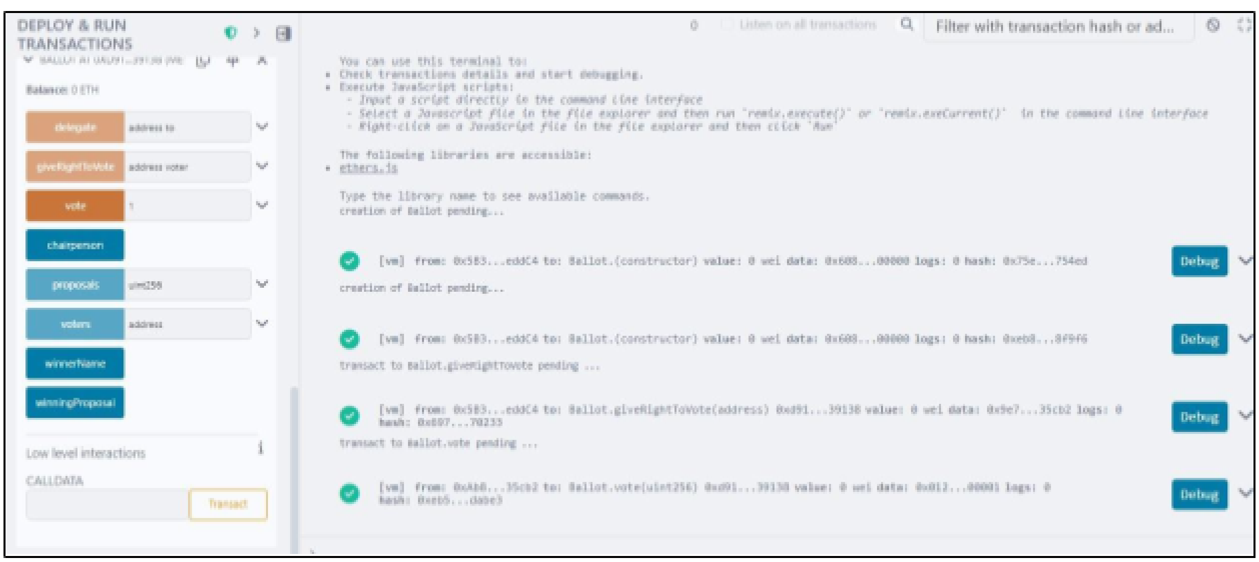


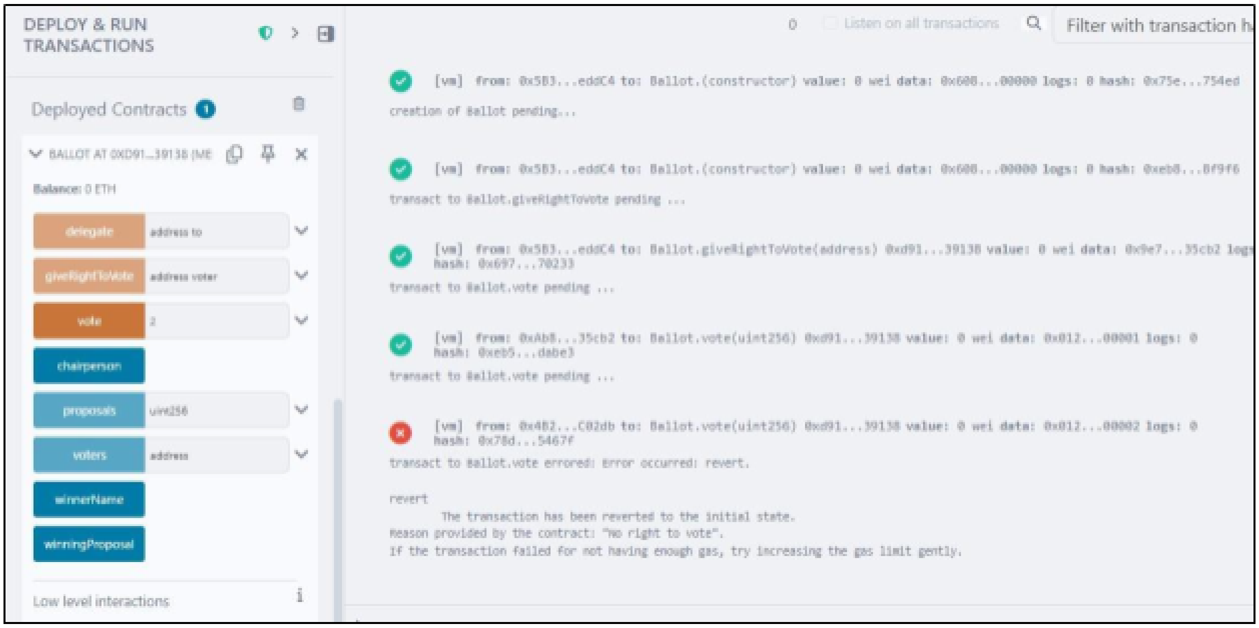
- Deploying and running of the contract



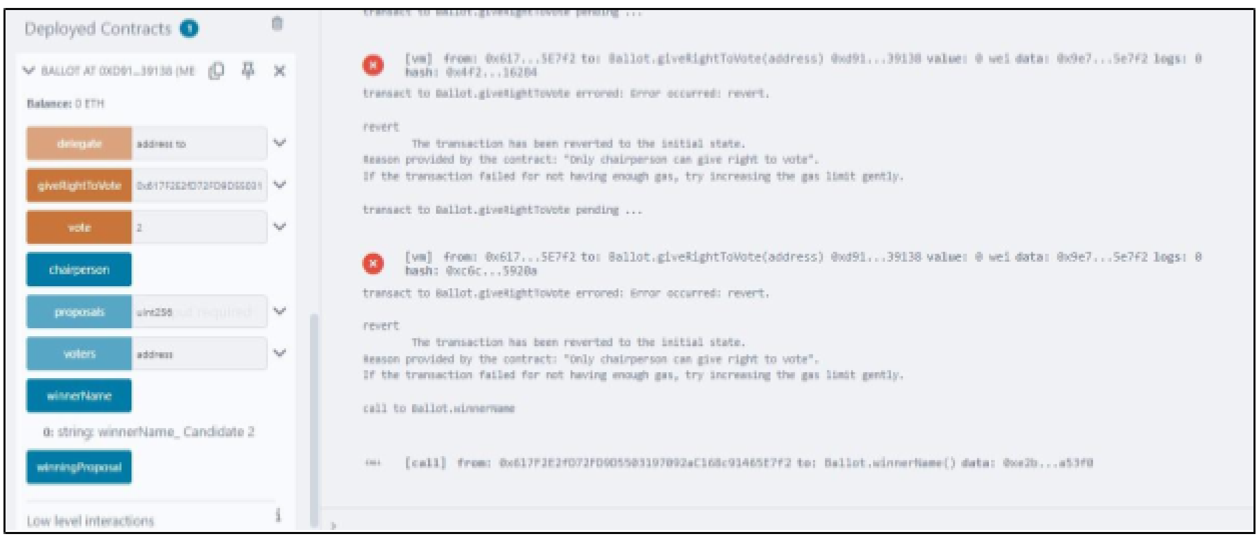
- Loading the Proposal Candidate’s Names (string)

Voted successfully:

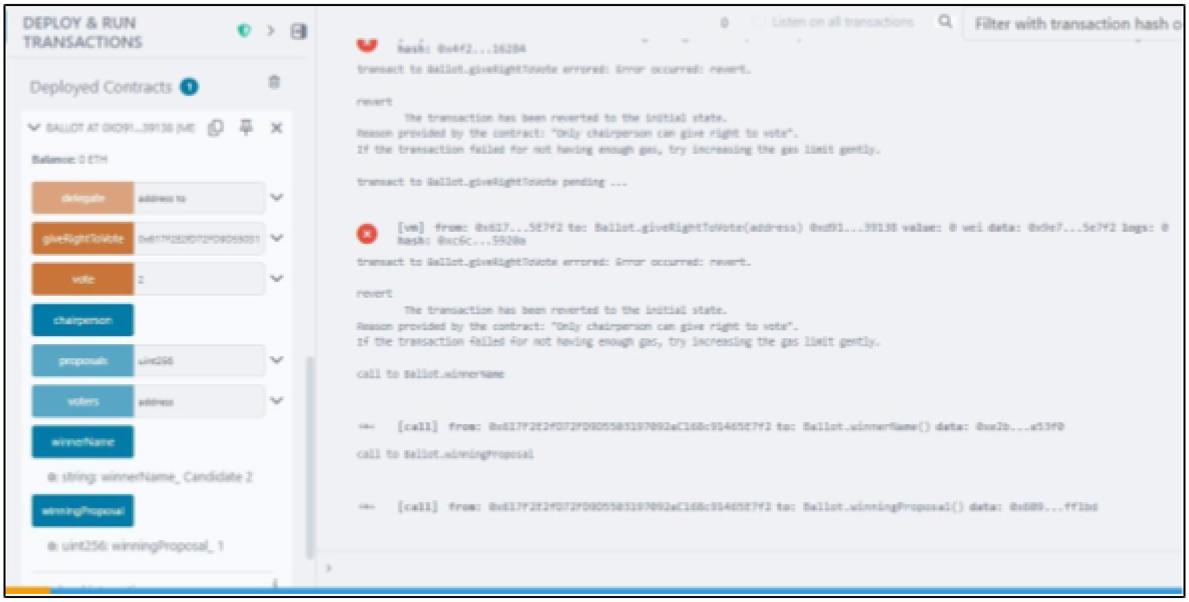




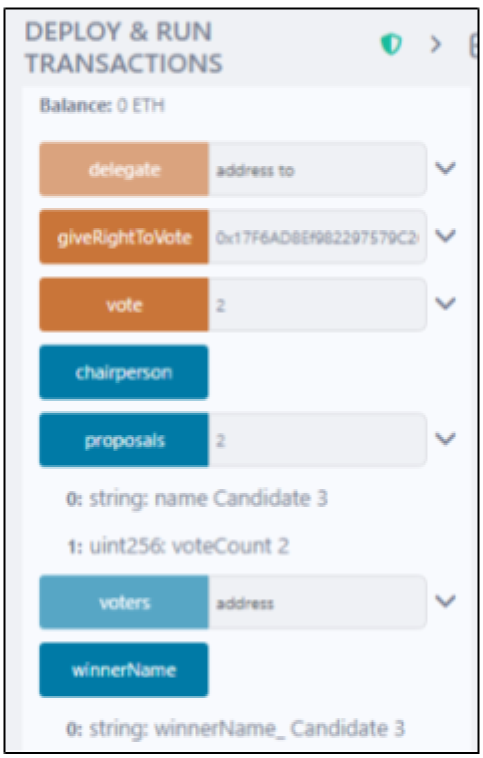
Tried to Vote twice which gave an error:



Voted from another node:



Checked the Winner node and the Proposals:



Conclusion

In this experiment, a Voting (Ballot) smart contract was implemented and deployed using Solidity in the Remix IDE environment. The practical use of ‘require’ statements demonstrated how smart contracts enforce validation rules and prevent unauthorized or incorrect actions. Core Solidity concepts such as mapping, storage, and memory were examined to understand how data is managed efficiently on the blockchain.